Mbrane

Final Report

Logic Programing

Group Mbrane 4:
Leonardo Fernandes Moura - 201706907@fe.up.pt
João Pedro Campos - up201704982@fe.up.pt

Faculdade de Engenharia da Universidade do Porto
Rua Roberto Frias, sn, 4200-465 Porto, Portugal

November 13, 2019

Summary

The goal of this project was to program a board game using the prolog language. In our case the game in question is Mbrane, a game created by DukeZhou based on Sudoku and strategy games like chess. Section 2 clarifies all the aspects of this game.

In terms of programing, the goal of the project was to efficiently implement all the game logic using Prolog. This means we have to represent the core mechanics of the game using Prolog. The first step is to find a way to represent the game board that's both efficient and relatively simple to program and operate. Then our goal is to make the game playable, continuously updating the state of the board as both players make their moves. In order to termiate the game and find the winner, we need to detect when the board is in a completed state, then we calculate the winner and present it. After the game is playable, the final step is to make it possible to play against the computer programing a bot that would be able to play in two difficulty settings. The first one (the easy mode) simply makes random plays, the second one calculates the best play it can make at the moment (greedy approach). This means we end up with three different game modes: player versus player, player versus bot and bot versus bot.

Besides all the game logic, the player needs to interact with the game, so we need to develop a simple text base UI, using Prolog predicates that are able to interact with the terminal input/output stream. This allows us to read inputs from the player (or both players) so that we can use them make moves.

The prolog distribution used was SicStus Prolog using a Student License.

Contents

1 Introduction

This project was developed during the Logic Programing Course in the context of the third year of the Integrated Masters in Informatics and Computing Engineering at FEUP. The goal was to implement the logic of a board game in Prolog, in this case *Mbrane*.

Here is a short explanation of all the sections in this report:

- **Game presentation:** Clarification of the most relevant aspets of the game, both it's rules and origin.
  - **Origin:** Description the origin of the game, and it was based on the journals of the game creator himself.
  - **Rules:** Description of all the rules of the game.
- **Game Logic:** Description of the game logic and how it was implememted in Prolog, it focuses on the most relevant predicates.
  - **Game state representation:** Description of how all the different states of the game are represented internally in Prolog.
  - **Board viewing:** Description of how we use the internal game state representation to display the game in a relatively user friendly way, using a text interface.
  - **List valid plays:** Description of how the we are able to determine if a play is valid or not, and more importantly, how we can find all the plays that can be made in a given moment.
  - **Game finalization:** Description of how we are able to detect the game is in a final state in order to determine the winner.
  - **Board Evaliation:** Description of how we can evaluate the board, in order to find if a given play is valid or not.
  - **Plays of the computer:** Description of the computer choses a move when given a list of possible moves.

2 Game Presentation

2.1 Origin

Mbrane is a fairly recent game. It was created in 2013 by someone who goes by the name of DukeZhou. When he was a kid, DukeZhou really liked to play strategy board games, but he didn't like games where luck was a relevant factor, he liked games where the player with superior strategy or the one "who played better" was definitely the winner. Games like chess or checkers were all he was into. n 2005 he discovered Sudoku, and even though puzzles were not his favorite type of game, he was amazed by the game. The sheer amount of combinations for a 9x9 board is around $6.7 * 10^21$. That factor combined with the all restraints the puzzle imposes made it feel somewhat magical.

In May 2013 while he was playing Fallout 3 the rules of Mbrane dropped into his head all at once. He immediately called a friend whom he used to play chess with and they spent hours playing and discovering the game, getting more and more fascinated with it as time went by.

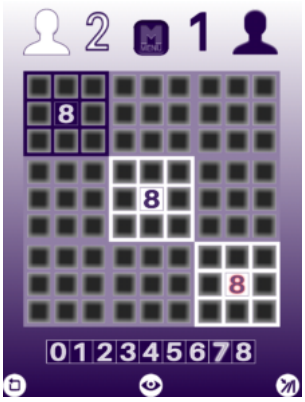
The full story, taken from the journals of DukeZhou can be read in the following address: http://mbranegame.com/origin-of-mbrane [DukeZhou, 2019a].

2.2 Rules

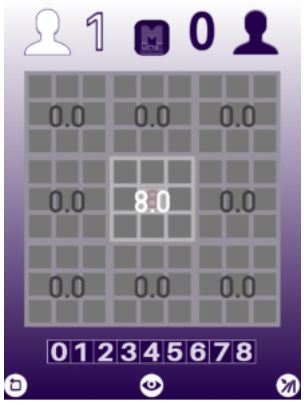
The following rules are the official rules for Mbrane, in their integral redaction. Mbrane is divided in two distinct phases: the placement phase and the resolution phase

2.2.1 Placement phase

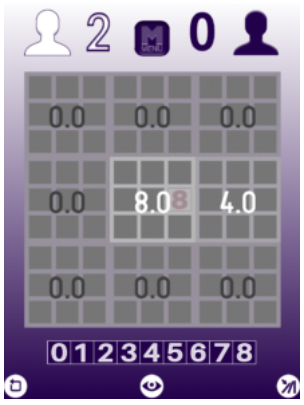
- Players take turns placing numbers in an empty Sudoku. These are bids for territories.



- Players receive points equal to the number in the region where the number is placed. These points are known as **Power**.



- If a number borders other regions, horizontally, vertically or diagonally, the player receives 1/2 points in those border regions. These points are known as **Influence**.

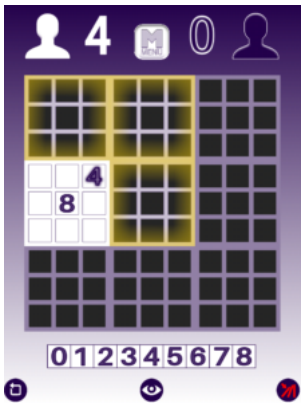


- Placement conforms to the rules of Sudoku, so a number may only be placed once in a region, row or column. (Broken Sudoku will occur.)
- When all numbers that can be placed have been placed, the gameboard is resolved. **Resolution** determines the outcome.

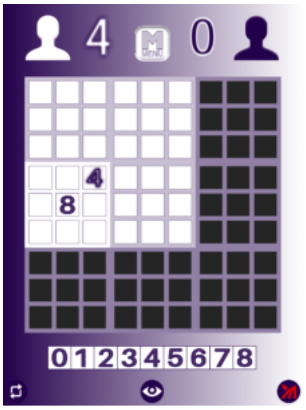
2.2.2 Resolution Phase

- Regions are resolved in order of the greatest disparity. These are the regions with greatest point difference between players.

- When a region is resolved, it is awarded to the player with the most points in the region, known as the dominant player.
- All opposing numbers in that region defect, flipping to the dominant player. This player is now said to have control.



- If these numbers border other regions, they switch their **Influence** points. (This can shift the balance of power, affecting the outcome of the game!)



- When all regions that can be resolved, have been resolved, the player controlling the most regions wins.
- Strength of victory is measured by the ratio of controlled regions between players, such as 5/4, 4/5, 4/4, 9/0, 0/9,...

The official rules can be consulted in the address: http://mbranegame.com/rules-of-m [DukeZhou, 2019b].

3 Game Logic

3.1 Game state representation

Although the user can only see one board at a time, in a any phase of the game, there are actually three boards represented in the game's data structures. There's a board predicate that represents the game **board** itself, a list of 9 lists each representing a line of the board. This is the board used to print the state of the game. There's also a **board_blocks** predicate. This predicate is also a list of 9 lists, each one representing a 3 by 3 block of the board, this is used to facilitate calculations involving power in each block and influence in adjacent blocks caused by the placement of a number in the edge of a block. Finally we have a **board_influence** list that stores the values of power (and spread influence) in each block of the board. When a player adds a number to cell, all the boards are updated accordingly. The predicate used to declare a fresh board is *start_board(-Board)*, which unifies a variable to a list of containing the three boards mentioned above. In order to distinguish both players we determined that Player1 would place positive numbers on the board, and Player2 would place negative numbers. This eases the calculations and makes it possible to display to the user the which player has controll in each block of the board

```
% get starting boards
start_board([BL,BB,BI]) :- board(BL), board_blocks(BB), board_influence(BI).
```

3.2 Board viewing

Before each move, the board is displayed on the screen. To display the board we use the predicate *display_game(+Board, +Player1, +Player2)*. This predicate first displays the players' names and then uses *display_board(+Board)* to display the board itself. The board is displayed line by line with a series of dividers. Although for game logic the values of the board can be positive or negative, the board is printed with only positive numbers. The power of all the blocks is also displayed using*display_power(+Power)*

```
display_board(B) :-
    write('     0   1   2   3   4   5   6   7   8  '), nl,
    display_board(B, 8).
display_board([L|T], I) :-
    I >= 0,
    write('   +---+---+---+---+---+---+---+---+---+'), nl,
    N is 8 - I, write(' '), write(N), write(' '),
    draw_line(L), nl,
    NI is I - 1,
    display_board(T, NI).
display_board(_, -1) :- write('   +---+---+---+---+---+---+---+---+---+'), nl.

draw_line(L) :- draw_line(L, 9).
draw_line(_, 0) :- write('|').
draw_line([H|T], J) :-
    J > 0, (H = ' ', V = ' '; abs(H,V)),
    write('| '), write(V), write(' '),
    NJ is J - 1,
    draw_line(T, NJ).


display_power(P) :- nl,write('Power '),write(P),nl,nl.
```

3.3 List valid plays

As per requested, we developed a predicate that returns a list of all possible moves. This predicate is *valid_moves(+Board, -R)*. The first argument of the

predicate is a list with the board of lines and board of blocks. This predicate works by first finding a list of all empty cells with *check_all(+Board, -List),*. Then it tests all possible values in those cells with *test_all(+List, +Board, +Result)*. *R* is a list of all the possible moves which are also lists in the format *[X,Y,Value]*.

```
test_all_cells(_,[], [BL,BB], Acc, Acc).

test_all_cells(0, [[X,Y]|T], [BL,BB], Acc, R) :-
    check_move(X, Y, 0, BL, BB),
    H = [X,Y,0],
    test_all_cells(8,T,[BL,BB],[H|Acc],R).

test_all_cells(0, [[X,Y]|T], [BL,BB], Acc, R) :-
    test_all_cells(8,T,[BL,BB],Acc,R).

test_all_cells(V, [[X,Y]|T], [BL,BB], Acc, R) :-
    check_move(X, Y, V, BL, BB),
    H = [X,Y,V],
    NV is V - 1,
    test_all_cells(NV, [[X,Y]|T], [BL,BB], [H|Acc],R).

test_all_cells(V,I,[BL,BB],Acc, R) :-
    NV is V - 1,
    test_all_cells(NV, I, [BL,BB], Acc,R).
```

3.4 Play execution

Each player can make a move per turn *player_turn(+Board, +Player, -NewBoard)*. In this turn predicate, the program asks the player what he wants to play with *get_move(-X, -Y, -V)* then the program tries that move with *move(+Move, +Board, -NewBoard)*. If the move is invalid the program repeats the previous action, otherwise, it returns a new board with the new piece.

```
% player_turn
% get player input and make move
player_turn([BL,BB,BI], P, R) :-
    repeat,
    get_move(X, Y, V),
    TV is P * V,
    check_move(X, Y, TV, BL, BB),!,
    move([X, Y, TV], [BL,BB,BI], R).
```

3.5 Game finalization

The game is in the final state if no more numbers can be placed on the board, according to placing rule already mentioned. We know this using the *valid_moves(+Board, -R)*. If the Result list is empty, then no more valid moves can be made, which means the game is in the final state. When this happens the *game_over(+Board, -Winner)* predicate calculates the winner of the game.

```
% game_over
game_over([B,BI],W) :-
    valid_moves(B, R),
    R = [],
    get_winner(BI,W).

% get_winner
get_winner(B,W) :-
    get_winner_(B,P1,P2),
    (P1 > P2, W is 1); W is -1.

get_winner_([],P1,P2) :- P1 is 0, P2 is 0.
get_winner_([H|T],P1,P2):-
    get_winner_(T,N1,N2),
    ((H > 0, P1 is N1 + 1, P2 is N2);
        (H < 0, P1 is N1, P2 is N2 + 1)).
```

3.6 Board evaluation

The number placing rules are the ones of Sudoku, so the player can only place a number if that same number is not present in either the column, row or 3x3 section of the square the player wants to place it. If these conditions are met, the number is able to be placed.

We evaluate if a play is possible using the *check_move(+X,+Y,+Value,+BL,+BB)*, it calculates if the move is possible using all the restrictions merntioned above, if some of those restrictions are not met the predicate fails indicating the move can't be made.

```
% check_move
check_move(X, Y, V, BL, BB) :-
    check_x(X), check_y(Y), check_v(V),
    check_cell(X, Y, BL), \+ check_restrictions(X, Y, V, BL, BB).
check_move(_,_,_,_,_) :- fail.
```

3.7 Plays of the computer

The computer plays only in an easy difficulty setting, this means that for each turn of the computer we get all the possible plays using the *valid_moves(+Board, -R)* predicate. Then using the *random/3* predicate from the SicStus random library we chose a random move and execute it. The predicate that makes the move move of the computer is *bot_turn(+Board,+Power,-Result)*.

```
bot_turn([BL,BB,BI], P, R) :-
    random_move([BL,BB],X,Y,V),
    TV is P * V,
    make_move([X, Y], TV, [BL,BB,BI], R).
```

4 Conclusions

Overall, the project was a really good way to practice the Prolog programming language, and also see how it can be used to create a real application (in this case a game). In the begginning we really struggled to understand the implications of the language and how we could use it to build a game. The total change of paradigm from what we were used to was really tough and the learning curve was a really rough one. Our start was very slow and we took some time before we could start working at a decent pace.

We had a lot of trouble during the development of the project. Everything started on our first approach to internally store and use the game board. Only when we were trying to make the game "playable" did we realize our approach was not the best one, as we were not finding a way to make the game play past the second round. This required an almost total refactoring of the code which made us loose a lot of time. We ended up not having time to implement a more difficult game mode agains the computer using a little aritifitial inteligence wich was something we were looking forward to do. Due to these time issues we also ended up not implementing the final *Resolution Phase*. We made this decision having in mind it whould not add much complexity to the project as it is only a bunch of calculations.

In the end of the development of the project we actually were programming at a good rythm. So we find this was a positive experience, that helped us getting a better understanding of declarative programming languages and logic programming.

Apendix

[DukeZhou, 2019a] DukeZhou (2019a). Origin of [m]. Available in `http://mbranegame.com/origin-of-mbrane`. Consulted in november 2019.

[DukeZhou, 2019b] DukeZhou (2019b). Rules of [m]. Available in `http://mbranegame.com/rules-of-m`. Consulted in november 2019.